Pytest Fixtures

Pedro Paulo Monteiro Muniz Barbosa

Estrutura 3A de testes:

1. **Arrange** - Criar alguns objetos
2. **Act** - Simular
3. **Assert** - Checar os resultados

Test-Driven Development By Example, Kent Beck - Pg. 99

Estrutura 3A de testes:

```python

def example_1(num):
    return 1

```

```python

def test_example():
    parameter = 1                   # -> Arrange
    expected = 1                    # -> Arrange
    result = example_1(parameter)   # -> Act
    assert result == expected       # -> Assert

```

Estrutura 3A de testes: One Step Test

```
12  def test_example():
13      assert example_1(1) == 1       # -> Arrange, Act, Assert
14
```

"Se quisermos remover duplicação de nosso código modelo, queremos removê-la do nosso código de teste também? Talvez."

Kent Beck, Test Driven-Development By Example - Pg. 163

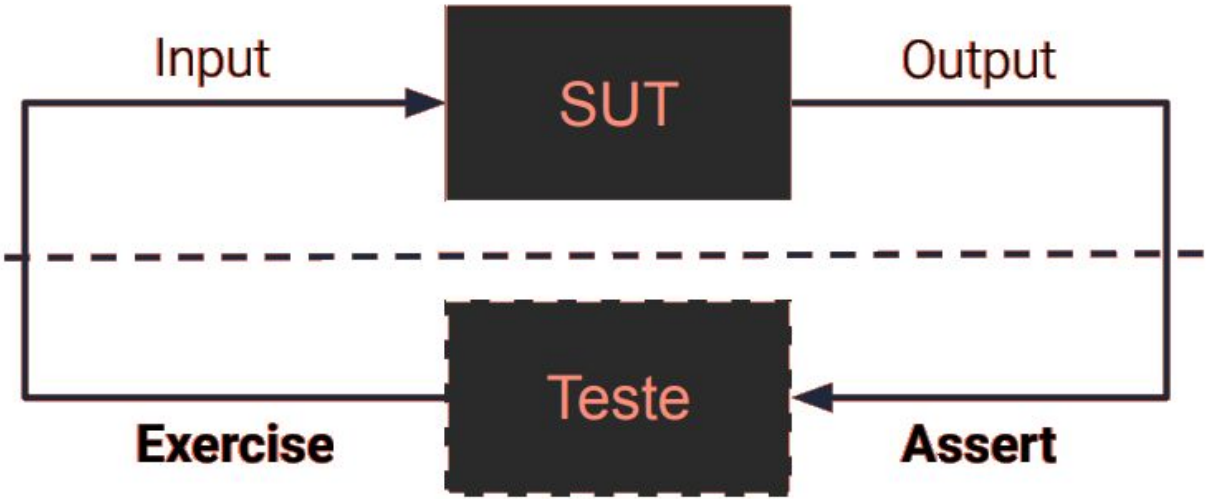
O que é uma Fixture?

"Fixtures" são funções que serão executadas antes de cada função de teste à qual é aplicada. "Fixtures" são usadas para alimentar alguns dados para os testes, como conexões de banco de dados, URLs para testar e algum tipo de dado de entrada. Portanto, em vez de executar o mesmo código para todos os testes, podemos anexar a função fixture aos testes e ela rodará e retornará os dados para o teste antes de executar cada um deles.

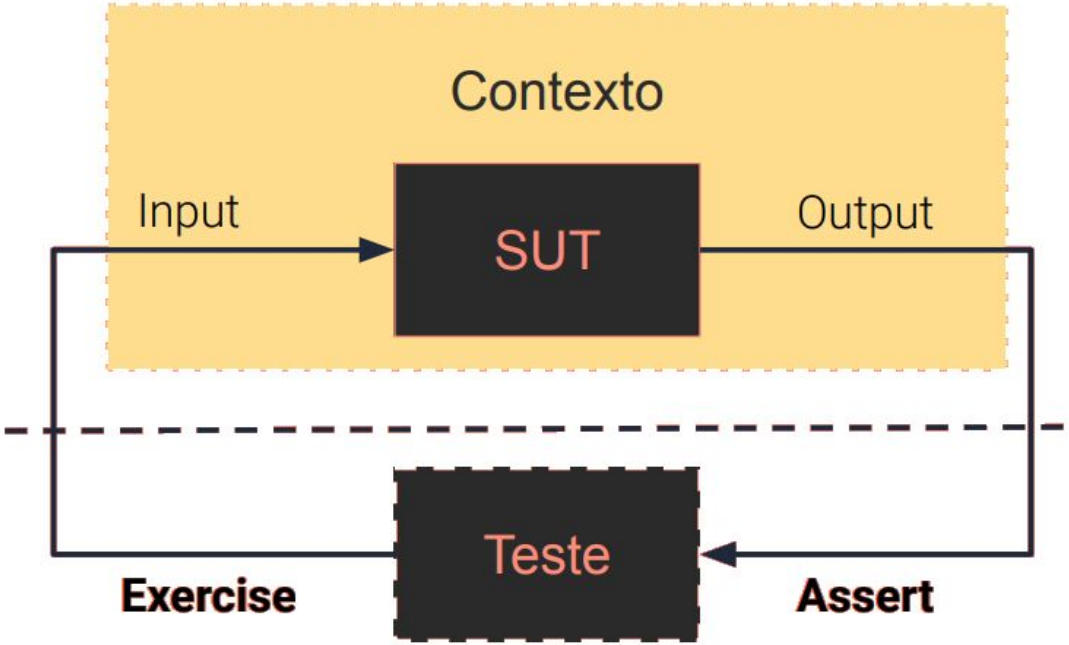
Four-Phases Test (Meszaros):

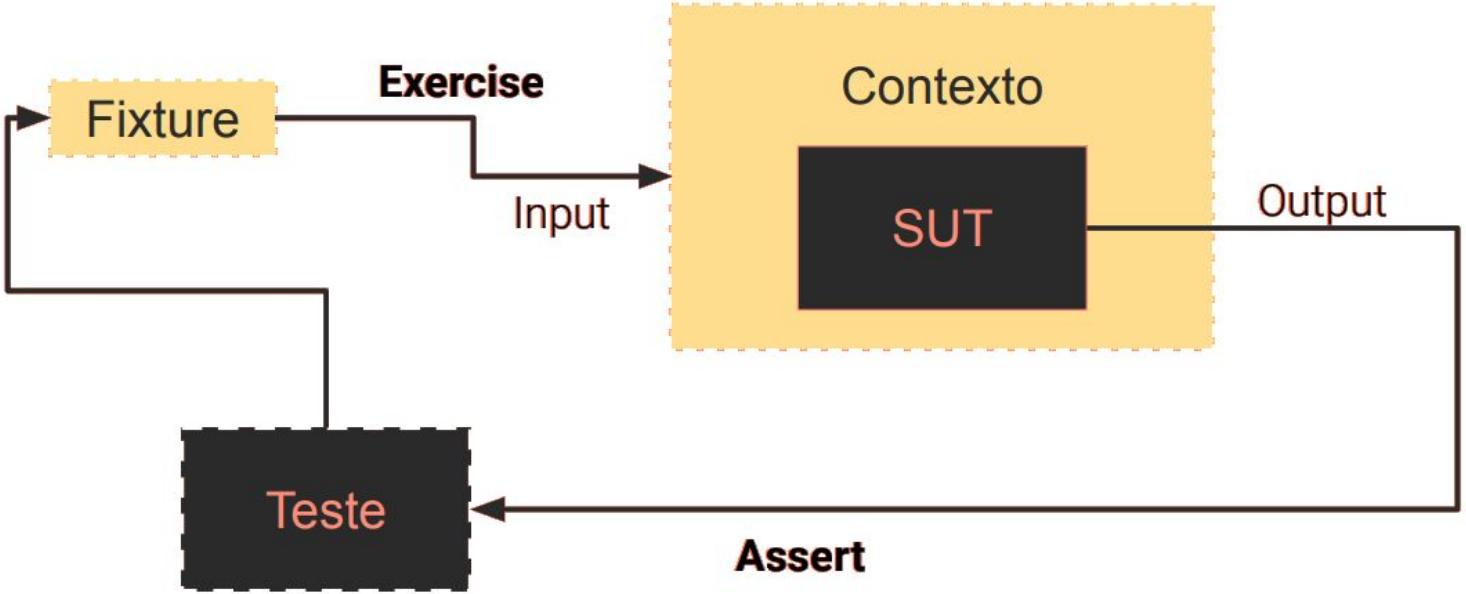
1. **Setup:** Na primeira fase, montamos o equipamento de teste que é necessário para que o SUT exiba o comportamento esperado, bem como qualquer coisa que você precise implementar para poder observar o resultado real.
2. **Exercise:** Na segunda fase, interagimos com o SUT.
3. **Verify:** Na terceira fase, fazemos o que for necessário para determinar se o resultado esperado foi obtido.
4. **TearDown:** Na quarta fase, desmontamos o dispositivo de teste para colocar o mundo de volta no estado em que o encontramos.

Testes usuais:



Testes utilizando fixtures:




Fixture
Exercise
Input
Contexto
SUT
Output
Teste
Assert

```python
import pytest


class Fruit:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name


@pytest.fixture
def my_fruit():
    return Fruit("apple")


@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]


def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

```
============================================================================ 1 passed in 0.17s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 1 item

tests_example_fixture.py::test_my_fruit_in_basket PASSED

============================================================================ 1 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```

Autouse:

Às vezes você pode querer ter uma fixture (ou mesmo várias) da qual você sabe que todos os seus testes vão depender. As fixtures de “uso automático” são uma maneira conveniente de fazer com que todos os testes as solicitem automaticamente. Isso pode eliminar muitas solicitações redundantes e pode até fornecer um uso mais avançado de fixtures.

Podemos tornar uma fixture uma fixture de uso automático passando autouse=True para o decorador da mesma.

```python
import pytest


@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order(first_entry):
    return []


@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]


def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

```
============================================================================ 2 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 2 items

tests_example_fixture.py::test_string_only ['a']
PASSED
tests_example_fixture.py::test_string_and_int ['a', 2]
PASSED

============================================================================ 2 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```

TearDown:

Quando executamos nossos testes, queremos garantir que eles se limpem para que não mexam com nenhum outro teste (e também para que não deixemos para trás uma montanha de dados de teste para sobrecarregar o sistema). As fixtures no pytest oferecem um sistema de desmontagem muito útil, que nos permite definir as etapas específicas necessárias para que cada instalação seja limpa depois de si mesma.

TearDown - yield:

As fixtures “yield” rendem em vez de retornar. Com essas fixtures, podemos executar algum código e passar um objeto de volta para o fixture/test solicitante, assim como com as outras fixtures. As únicas diferenças são:

- o *return* é trocado pelo *yield*.
- Qualquer código de desmontagem para essa fixture é colocado após o *yield*.

Assim que o pytest descobrir uma ordem linear para as fixtures, ele executará cada um até retornar ou render, e então passar para o próximo fixture na lista para fazer a mesma coisa.

Assim que o teste terminar, pytest voltará na lista de fixtures, mas na ordem inversa, pegando cada um que rendeu e executando o código dentro dele que estava **após** a instrução yield.

Como um exemplo simples, digamos que queremos testar o envio de e-mail de um usuário para outro. Teremos que primeiro fazer cada usuário, depois enviar o e-mail de um usuário para o outro e, finalmente, afirmar que o outro usuário recebeu essa mensagem em sua caixa de entrada. Se quisermos limpar após a execução do teste, provavelmente teremos que garantir que a caixa de correio do outro usuário seja esvaziada antes de excluir esse usuário, caso contrário, o sistema poderá reclamar.

```python
class MailAdminClient:
    def create_user(self):
        return MailUser()

    def delete_user(self, user):
        # do some cleanup
        pass

class MailUser:
    def __init__(self):
        self.inbox = []

    def send_email(self, email, other):
        other.inbox.append(email)

    def clear_mailbox(self):
        self.inbox.clear()


class Email:
    def __init__(self, subject, body):
        self.subject = subject
        self.body = body
```

```python
import pytest
from emaillib import Email, MailAdminClient

@pytest.fixture
def mail_admin():
    return MailAdminClient()


@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)


@pytest.fixture
def receiving_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)


def test_email_received(receiving_user, sending_user):
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(email, receiving_user)
    assert email in receiving_user.inbox
```

```
============================================================================ 1 passed in 0.18s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 1 item

tests_example_fixture.py::test_email_received
Creating the receiving user
<emaillib.MailUser object at 0x000002DB2F8A2D10>
Creating the sending user
<emaillib.MailUser object at 0x000002DB2F928340>
PASSED
Deleting the sending user
Deleting the receiving user
```

Fixtures Scopes:

As fixtures são criadas quando solicitadas pela primeira vez por um teste e são destruídas com base em seu escopo:

- `function`: o escopo padrão, a fixture é destruída ao final do teste.
- `class`: a fixture é destruída durante o teardown do último teste da classe.
- `module`: a fixture é destruída durante o teardown do último teste no módulo.
- `package`: a fixture é destruída durante o teardown do último teste no pacote.
- `session`: a fixture é destruída no final da sessão de teste.

Disponibilidade da Fixture:

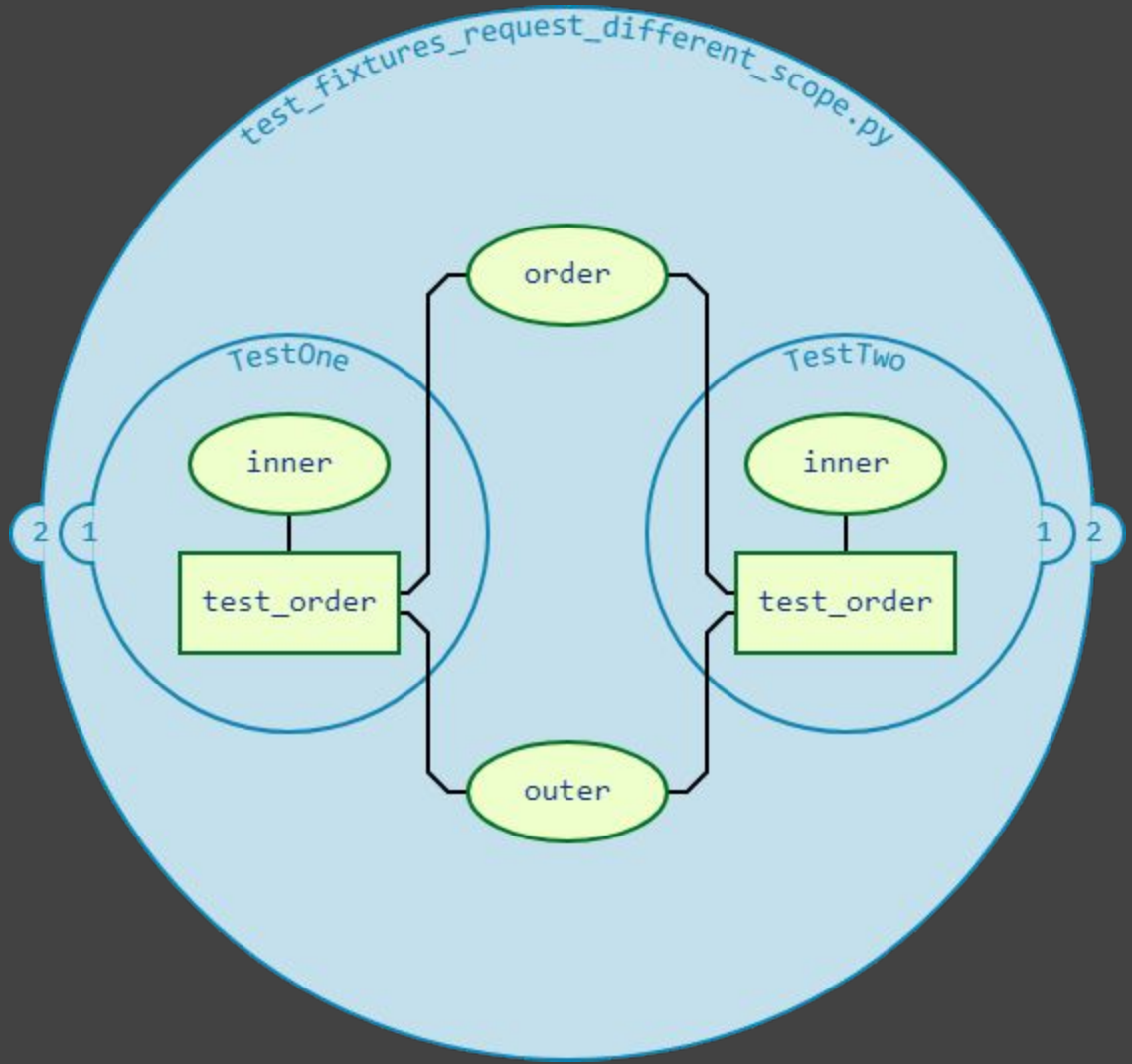
A disponibilidade da fixture é determinada a partir da perspectiva do teste. Um fixture só está disponível para testes a serem solicitados se estiverem no escopo em que o fixture está definido. Se um fixture for definido dentro de uma classe, ele só pode ser solicitado por testes dentro dessa classe. Mas se um fixture for definido dentro do escopo global do módulo, então cada teste naquele módulo, mesmo se estiver definido dentro de uma classe, pode solicitá-lo.

Da mesma forma, um teste também só pode ser afetado por uma fixture de uso automático se esse teste estiver no mesmo escopo em que a fixture de uso automático está definida.

Uma fixture também pode solicitar qualquer outra fixture, não importa onde esteja definido, desde que o teste que os solicita possa ver todos os fixtures envolvidos.

```python
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def outer(order, inner):
    order.append("outer")

class TestOne:
    @pytest.fixture
    def inner(self, order):
        order.append("one")

    def test_order(self, order, outer):
        assert order == ["one", "outer"]

class TestTwo:
    @pytest.fixture
    def inner(self, order):
        order.append("two")

    def test_order(self, order, outer):
        assert order == ["two", "outer"]
```

```
============================================================================ 2 passed in 0.15s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 2 items

tests_example_fixture.py::TestOne::test_order PASSED
tests_example_fixture.py::TestTwo::test_order PASSED

============================================================================ 2 passed in 0.15s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```


test_fixtures_request_different_scope.py
order
TestOne
inner
2
1
test_order
TestTwo
inner
1
2
test_order
outer

As Fixtures são definidas usando o decorador *@pytest.fixture*. O Pytest possui built-in fixtures úteis, listadas aqui para referência:

capfd
capfdbinary
caplog
capsys
capsysbinary
cache
doctest_namespace
monkeypatch
pytestconfig
record_property
record_testsuite_property
recwarn
request
testdir
tmp_path
tmp_path_factory
tmpdir
tmpdir_factory

Referências:

MESZAROS, Gerard. *xUnit Test Patterns*. Addison-Wesley, 2007.

BECK, Kent. *Test Driven-Development By Example*. Addison-Wesley, 2000.

pytest fixtures: explicit, modular, scalable. pytest, Full pytest Documentarion, 2015. Disponível em: <https://docs.pytest.org/en/6.2.x/fixture.html#what-fixtures-are>. Acesso em: 02 de fevereiro de 2022.